CPU Scheduling

key concepts

round robin, shortest job first, MLFQ, multi-core scheduling, cache affinity, load balancing

reading

Three Easy Pieces: Chapter 7 (CPU Scheduling), Chapter 8 (Multi-level Feedback), Chapter 10 (Multi-CPU Scheduling)

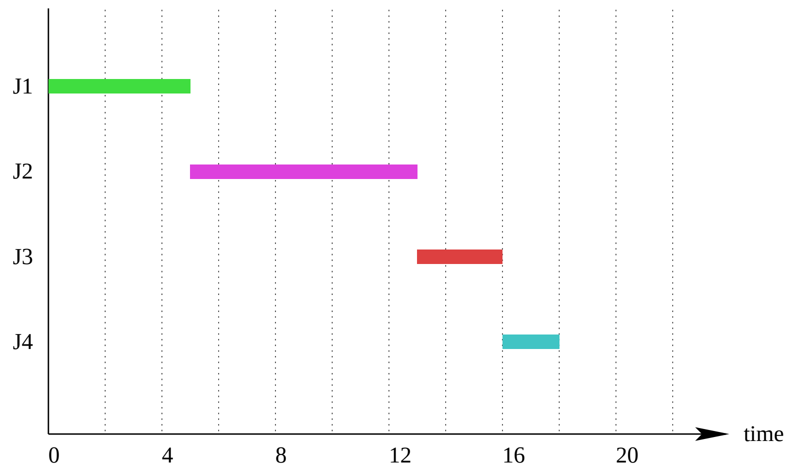
Simple Scheduling Model

- We are given a set of *jobs* to schedule.
- Only one job can run at a time.
- For each job, we are given
  - job arrival time (a_i)
  - job run time (r_i)
- For each job, we define
  - response time: time between the job's arrival and when the job starts to run
  - turnaround time: time between the job's arrival and when the job finishes running.
- We must decide when each job should run, to achieve some goal, e.g., minimize average turnaround time, or minimize average response time.

Basic Non-Preemptive Schedulers: FCFS and SJF

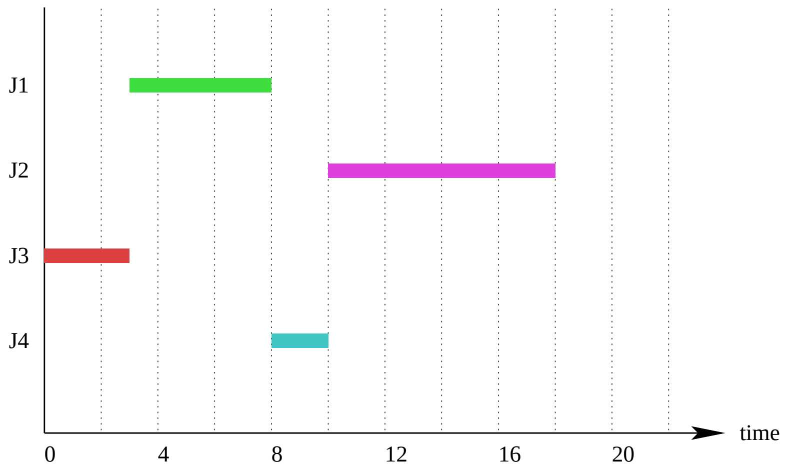
- FCFS: runs jobs in arrival time order.
  - simple, avoids starvation
  - pre-emptive variant: round-robin
- SJF: shortest job first - run jobs in increasing order of r_i
  - minimizes average *turnaround* time
  - long jobs may starve
  - pre-emptive variant: SRTF (shortest remaining time first)

FCFS Gantt Chart Example



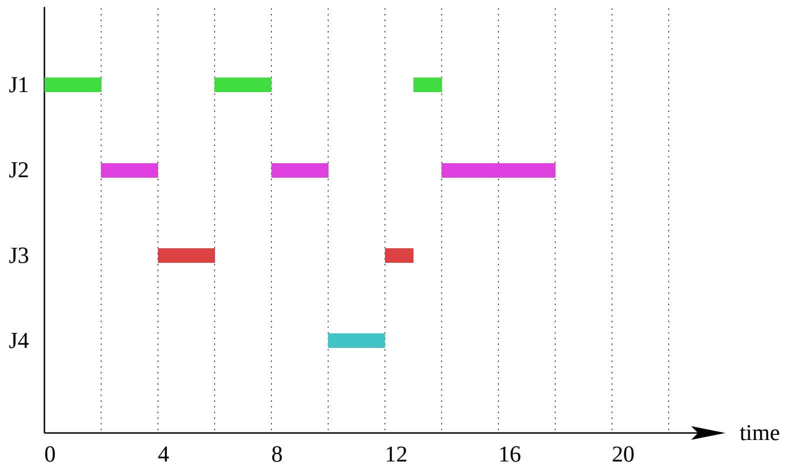
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SJF Example



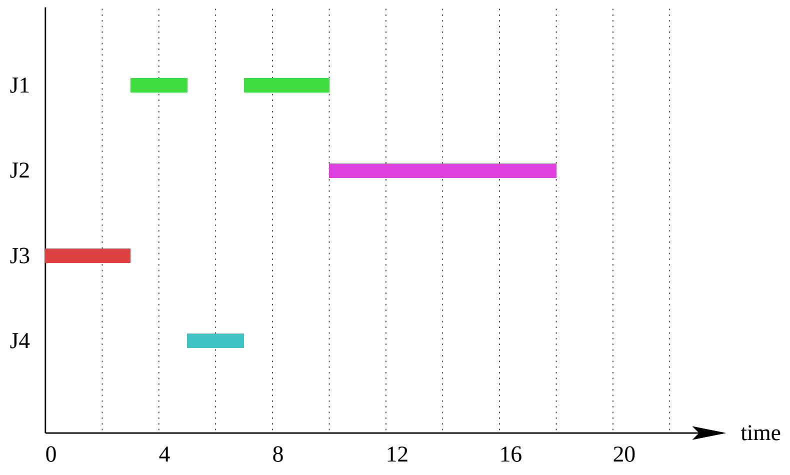
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

Round Robin Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SRTF Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

CPU Scheduling

- In CPU scheduling, the “jobs” to be scheduled are the threads.
- CPU scheduling typically differs from the simple scheduling model:
  - the run times of threads are normally not known
  - threads are sometimes not runnable: when they are blocked
  - threads may have different priorities
- The objective of the scheduler is normally to achieve a balance between
  - responsiveness (ensure that threads get to run regularly),
  - fairness,
  - efficiency

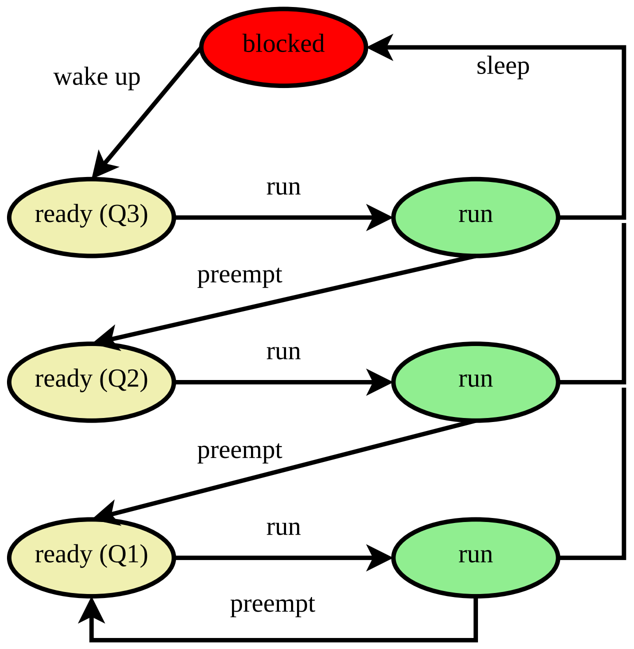
Multi-level Feedback Queues

- objective: good responsiveness for *interactive* threads, non-interactive threads make as much progress as possible
  - key idea: interactive threads are frequently blocked
- approach: given higher priority to interactive threads, so that they run whenever they are ready.
- problem: how to determine which threads are interactive and which are not?

Multi-level Feedback Queues (Algorithm)

- scheduler maintains n round-robin ready queues ($Q_1 \ldots Q_n$)
- scheduler always chooses a thread from Q_n, unless it is empty
  - if Q_n is empty, choose a thread from Q_{n-1}, unless it is empty too
  - and so on, choosing a thread from Q_1 only if all other queues are empty.
- threads in queue Q_i use quantum q_i
  - typically larger quanta for lower-priority threads ($q_i \geq q_{i+1}$)
- if the running thread from Q_i uses its entire quantum and gets preempted, demote it to queue Q_{i-1}
- if a thread blocks, put it into Q_n when it wakes up
- to prevent starvation, periodically move all threads to Q_n

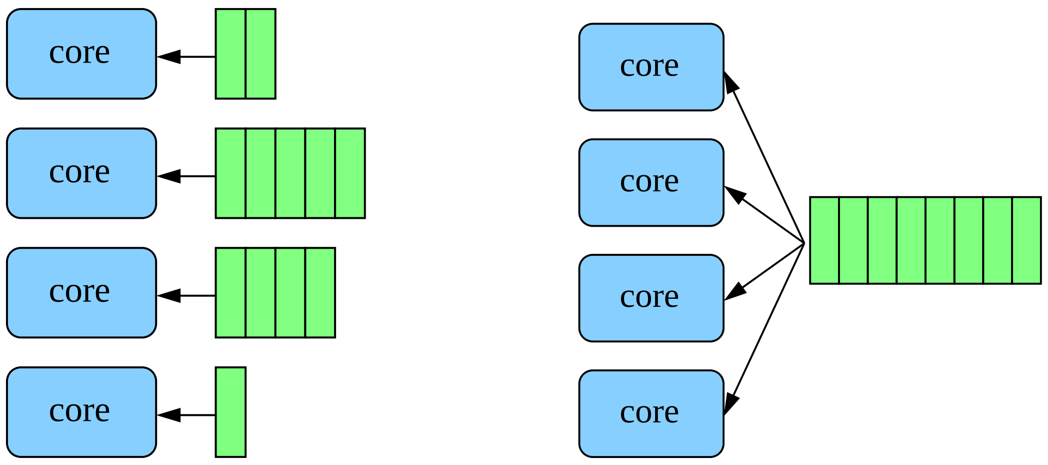
3 Level Feedback Queue State Diagram



Linux Completely Fair Scheduler (CFS) - Main Ideas

- each thread can be assigned a *weight*
- the goal of the scheduler is to ensure that each thread gets a "share" of the processor in proportion to its weight
- basic operation
  - track the "virtual" runtime of each runnable thread
  - always run the thread with the lowest virtual runtime
- virtual runtime is actual runtime adjusted by the thread weights
  - suppose w_i is the weight of the ith thread
  - actual runtime of ith thread is multiplied by $\frac{\sum_j w_j}{w_i}$
  - virtual runtime advances slowly for threads with high weights, quickly for threads with low weights

Scheduling on Multi-Core Processors



per core ready queue(s) vs. shared ready queue(s)

Scalability and Cache Affinity

- Contention and Scalability
  - access to shared ready queue is a critical section, mutual exclusion needed
  - as number of cores grows, contention for ready queue becomes a problem
  - per core design *scales* to a larger number of cores
- CPU cache affinity
  - as thread runs, data it accesses is loaded into CPU cache(s)
  - moving the thread to another core means data must be reloaded into that core's caches
  - as thread runs, it acquires an *affinity* for one core because of the cached data
  - per core design benefits from affinity by keeping threads on the same core
  - shared queue design does not

Load Balancing

- in per-core design, queues may have different lengths
- this results in *load imbalance* across the cores
  - cores may be idle while others are busy
  - threads on lightly loaded cores get more CPU time than threads on heavily loaded cores
- not an issue in shared queue design
- per-core designs typically need some mechanism for *thread migration* to address load imbalances
  - migration means moving threads from heavily loaded cores to lightly loaded cores